4.4 Repeated Execution: For Loops

Collections in Python can be used in many ways. We have already seen how we can use built-in aggregation functions (e.g., `any`, `all`, `max`) to perform computations across all elements of a collection (e.g., `list`, `set`).

But right now, we're limited by what aggregation functions Python makes available to us: for example, there's a built-in `sum` function, but no `product` function.[1] So in this section, we'll learn about the `for` loop, a compound statement that will allow us to implement our own custom aggregation functions across different types of collection data.

[1] That's not exactly true: there is a `math.product` function, but let's ignore that here. :)

Introducing the problem: repeating code

Suppose we wanted to write a function that computes the sum of a list of numbers, without using the built-in `sum` function.

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
```

If we knew the size of `numbers` in advance, we could write a single expression to do this. For example, here is how we could implement `my_sum` if we knew that `numbers` always contained three elements:

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
    return numbers[0] + numbers[1] + numbers[2]
```

But of course, this approach doesn't work for general lists, when we don't know ahead of time how many elements the input will have. We need a way to repeat the "`+ numbers[_]`" for an arbitrary number of list elements. Here is another way of writing our three-element code to pull out the exact statement that is repeated.

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
    sum_so_far = 0

    sum_so_far = sum_so_far + numbers[0]
    sum_so_far = sum_so_far + numbers[1]
    sum_so_far = sum_so_far + numbers[2]

    return sum_so_far
```

This implementation follows how a human might add up the numbers in the list. First, we start a counter a 0 (using a variable called `sum_so_far`). Then, we use three assignment statements to update the value of `sum_so_far` by adding another element of `numbers`. Let's look at the first such statement:

```
sum_so_far = sum_so_far + numbers[0]
```

This looks fairly straightforward, but is actually a big leap from the assignment statements we've studied before! What's unusual about it is that for the first time, we are assigning a value to a variable that has *already* been given a value. This type of assignment statement is called a **variable reassignment statement**. This statement is especially tricky because the variable `sum_so_far` appears on both sides of the `=`. We can make sense of this statement by reviewing the evaluation order that Python follows when executing an assignment statement:

- First, the right-hand side of the assignment statement (`sum_so_far + numbers[0]`) is evaluated.
- Second, the value produced by evaluating the right-hand side is stored in the variable on the left-hand side (`sum_so_far`).

We can visualize how the three assignment statements work by tracing through an example. Let's consider calling our doctest example, `my_sum([10, 20, 30])`. What happens to the value of `sum_so_far`?

Statement	**`sum_so_far` *after* executing statement**	**Notes**
`sum_so_far = 0`	`0`	
`sum_so_far = sum_so_far + numbers[0]`	`10` (`0 + 10`)	When evaluating the right-hand side, `sum_so_far` is `0` and `numbers[0]` is `10`.

Statement	`sum_so_far` *after* executing statement	Notes
`sum_so_far = sum_so_far + numbers[1]`	`30` (`10 + 20`)	When evaluating the right-hand side, `sum_so_far` is `10` and `numbers[1]` is `20`.
`sum_so_far = sum_so_far + numbers[2]`	`60` (`30 + 30`)	When evaluating the right-hand side, `sum_so_far` is `30` and `numbers[2]` is `30`.

Now that we understand this implementation, we can see that the statement `sum_so_far = sum_so_far + numbers[_]` is exactly what needs to be repeated for every element of the input list. So now, let's learn how to perform *repeated execution* of Python statements.

The for loop

In Python, the **for loop** is a compound statement that repeats a block of code once for element in a collection. Here is the syntax of a for loop:[2]

[2] Notice that the syntax is very similar to a comprehension. The key difference is that a comprehension evaluates an *expression* once for each element in a collection, but a for loop evaluates a *sequence of statements* once per element.

```
for <loop_variable> in <collection>:
    <body>
```

There are three parts:

1. `<collection>` is an expression for a Python collection (e.g., a `list` or `set`).
2. `<loop_variable>` is a name for the *loop variable* that will refer to an element in the colleciton.
3. `<body>` is a sequence of one or more statements that will be repeatedly executed. This is called the *body* of the for loop. The statements within the loop body may refer to the loop variable to access the "current" element in the collection.

   Just as we saw with if statements, the body of a for loop *must* be indented relative to the `for` keyword.

When a for loop is executed, the following happens:

1. The loop variable is assigned to the first element in the collection.
2. The loop body is executed, using the current value of the loop variable.

3. Steps 1 and 2 repeat for the second element of the collection, then the third, etc. until all elements of the collection have been assigned to the loop variable exactly once.

   Each individual execution of the loop body is called a **loop iteration**.

As with if statements, for loops are a control flow structure in Python because they modify the order in which statements are executed—in this case, by repeating a block of code multiple times. The reason we use the term *loop* is because after the last statement in the loop body is executed, the Python interpreter "loops back" to the beginning of the for loop, assigning the loop variable to the next element in the collection.

`my_sum` and the accumulator pattern

Now let us see how to use a for loop to implement `my_sum`. We left off with the following block of repeated code:

```
sum_so_far = sum_so_far + numbers[0]
sum_so_far = sum_so_far + numbers[1]
sum_so_far = sum_so_far + numbers[2]
```

We can now move the repeated `sum_so_far = sum_so_far + _` part into a `for` loop as follows:[3]

[3] Notice our loop variable name! A good convention to follow is that collections have a pluralized name (`numbers`), and loop variables have the singular version of that name (`number`).

```
for number in numbers:
    sum_so_far = sum_so_far + number
```

One important thing to note is that we no longer need to use list indexing (`numbers[_]`) to access individual list elements. The for loop in Python handles the extracing of individual elements for us, so that our loop body can focus just on what to do with each element.

With this, we can now write our complete implementation of `my_sum`.

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
    sum_so_far = 0

    for number in numbers:
```

```
        sum_so_far = sum_so_far + number

    return sum_so_far
```

Now, no matter how many elements `numbers` has, the loop body `sum_so_far = sum_so_far + number` will repeat once for each element. The ability to write a small amount of code that processes an arbitrary amount of data is one of the truly remarkable feats of computer science.

Accumulators and tracing through loops

Because of the variable reassignment, `sum_so_far` is more complex than every other variable we have used so far in this course. And because this reassignment happens inside the loop body, it happens once for each element in the collection, not just once or twice. This frequent reassignment can make loops hard to reason about, especially as our loop bodies grow more complex, and so we will take some time now to introduce a formal process you can use to reason about loops in your code.

First, some terminology. We call the variable `sum_so_far` the **loop accumulator**. The purpose of a loop accumulator is to store an aggregated result based on the elements of the collection that have been previously visited by the loop. In the case of `my_sum`, the loop accumulator `sum_so_far` stores, well, the sum of the elements that we have seen so far in the loop. We can keep track of the execution of the different iterations of the loop in a tracing table consisting of three columms: how many iterations have occurred so far, the value of the loop variable for that iteration, and the value of the loop accumulator at the *end* of that iteration. We call this table a **loop accumulation table**. Here is the loop accumulation table for a call to `my_sum([10, 20, 30])`:

Iteration	Loop variable (`number`)	Loop accumulator (`sum_so_far`)
0	N/A	0
1	10	10
2	20	30
3	30	60

Almost every for loop has an accumulator variable.[4] To distinguish these from other variables, we recommend using the `_so_far` suffix in the variable name, and optionally adding a comment in your code explaining the purpose of the variable.

[4] Later, some might even have more than one.

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the numbers in numbers.

    >>> my_sum([10, 20, 30])
```

```
    60
    """
    # ACCUMULATOR sum_so_far: keep track of the running sum of the elements
         in numbers.
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

When the collection is empty

What happens if we call `my_sum` on an empty list?

```
>>> my_sum([])
0
```

Why does this happen? The key to understanding this is that when we loop over an empty collection, zero iterations occur and the loop body never executes. So when we call `my_sum([])`, first `sum_so_far` is assigned to `0`, and then the for loop does not execute any code, and so `0` is returned. A key observation here is that *when the collection is empty, the initial value of `sum_so_far` is returned*.

Designing loops using the accumulator pattern

Our implementation of `my_sum` illustrates a more general pattern that we'll employ when we use loops to perform an aggregation computation. Here is the **accumulator pattern**:

1. Choose a meaningful name for an accumulator variable based on what you're computing. Use the suffix `_so_far` to remind yourself that this is an accumulator.
2. Pick an initial value for the accumulator. This value is usually what should be returned if the collection is empty.
3. In the loop body, update the accumulator variable based on the current value of the loop variable.
4. After the loop ends, return the accumulator.

Here is a *code template* to illustrate this pattern.

```
<x>_so_far = <default_value>

for element in <collection>:
    <x>_so_far = ... <x>_so_far ... element ...  # Somehow combine loop
         variable and accumulator

return <x>_so_far
```

Code templates are helpful when learning about programming techniques, as they give you a natural starting point in your code with "places to fill in". However, as we'll see over the next few sections, we should not blindly follow code templates either. Part of mastering a code template is deciding when to use it and when to modify it to solve the problem at hand.

Accumulating the product

Let's use the accumulator pattern to implement the function `product`:

```
def product(numbers: list[int]) -> int:
    """Return the product of the given numbers.

    >>> product([10, 20])
    200
    >>> product([-5, 4])
    -20
    """
    # ACCUMULATOR product_so_far: keep track of the product of the
    # elements in numbers seen so far in the loop.
    product_so_far = 1

    for number in numbers:
        product_so_far = product_so_far * number

    return product_so_far
```

Notice how similar the code for `product` is to `my_sum`. In fact, disregarding the changes in variable names, the only changes are:

- the initial value of the accumulator (`0` versus `1`)
- the "update" operation inside the loop body (`+` versus `*`)

Looping over sets

Because sets are collections, we can use for loops to iterate through the elements of a set as well. However, because sets are unordered, we cannot assume a particular order that the for loop will visit the elements in. So for loops over sets should only be used when *the same result would be obtained regardless of the order of the elements*. The aggregation functions we've looked at so far like `sum` satisfy this property.

Looping over strings

Strings are very similar to lists because they are considered ordered sequences of data. Python treats a string as an ordered collection of characters (strings of length one), and so

we can use for loops with strings to iterate over its characters one at a time.

Here is an example of using a for loop to count the number of characters in a string.

```
def my_len(s: str) -> int:
    """Return the number of characters in s.

    >>> my_len('David')
    5
    """
    # ACCUMULATOR len_so_far: keep track of the number of
    # characters in s seen so far in the loop.
    len_so_far = 0

    for character in s:
        len_so_far = len_so_far + 1

    return len_so_far
```

Unlike `my_sum`, here we do not use the loop variable to update the accumulator `len_so_far`. This is because we don't care what the actual value character is, we are only counting iterations. In these scenarios, we can use an underscore _ in place of the name for the loop variable to communicate that the loop variable is not used in the body of the for loop:

```
def my_len(s: str) -> int:
    """Return the number of characters in s.

    >>> my_len('David')
    5
    """
    # ACCUMULATOR len_so_far: keep track of the number of
    # characters in s seen so far in the loop.
    len_so_far = 0

    for _ in s:
        len_so_far = len_so_far + 1

    return len_so_far
```

Looping over dictionaries

Python dictionaries are also iterable. Just like we saw with comprehensions, when we iterate over a dictionary, the loop variable refers to the *key* of each key-value pair. But of course, we can use the key to lookup its corresponding value in the dictionary.

For example, suppose we are given a dictionary mapping restaurant menu items (as strings) to their prices (as floats). Here is how we could calculate the sum of all the prices

on the menu.

```
def total_menu_price(menu: dict[str, float]) -> float:
    """Return the total price of the given menu items.

    >>> total_menu_price({'fries': 3.5, 'hamburger': 6.5})
    10.0
    """
    # ACCUMULATOR total_so_far: keep track of the total cost of
    # all items in the menu seen so far in the loop.
    total_so_far = 0.0

    for item in menu:
        total_so_far = total_so_far + menu[item]

    return total_so_far
```

The loop variable `item` refers to the *keys* in the dictionary, so to access the corresponding prices we need to use a key lookup expression, `menu[item]`. Here is how we can visualize this using a loop accumulation table:

Iteration	Loop variable (`item`)	Loop accumulator (`total_so_far`)
0		`0.0`
1	`'fries'`	`6.5`
2	`'hamburger'`	`10.0`

One final note: like sets, dictionaries are unordered. We chose a particular order of keys for the loop accumulation table just to understand the loop behaviour, but we should not assume that this is the guaranteed order the keys would be visited. Just as with sets, only loop over dictionaries when your computation does *not* depend on the iteration order.

A new type annotation: `Iterable`

Something you might notice about the two functions `my_len` and `my_sum` we've developed so far is that actually work on more types than currently specified by their parameter type annotation. For example, `my_len` works just as well on lists, sets, and other collections. If we look at the function body, we don't use the fact that `s` is a string at all—just that it can be iterated over. It would be nice if we could relax our type contract to allow for any collection argument value.

We say that a Python data type is **iterable** when its values can be used as the "collection" of a for loop, and that a Python object is iterable when it is an instance of an iterable data type.[5] This is equivalent to when a value can be used as the "collection" of a

[5] You might wonder why Python doesn't just call these "collections" instead. There is a technical reason that is beyond the scope of this course, but for our purposes, we'll treat "iterable" and "collection" as synonymous.

comprehension. We can import the `Iterable` type from `typing` to indicate that a value must be any data type that is iterable. Here's how we would write a more general `my_len`:

```
from typing import Iterable


def my_len(collection: Iterable) -> int:
    """Return the number of elements in collection.

    >>> my_len('David')
    5
    >>> my_len([1, 2, 3])
    3
    >>> my_len({'a': 1000})
    1
    """
    len_so_far = 0

    for _ in collection:
        len_so_far = len_so_far + 1

    return len_so_far
```

Notice that other than renaming a variable, we did not change the function body at all! This demonstrates how powerful the accumulator pattern can be; accumulators can work with any iterable object.

Alternatives to for loops

You may feel that several of the examples in this section are contrived. You are not wrong; we are trying to leverage your familiarity with the built-in functions to help introduce a new concept. You may also have noticed that there are other ways to solve some of the problems we've presented. For example, `average_menu_price` can be solved using comprehensions rather than loops:

```
def average_menu_price_v2(menu: dict[str, float]) -> float:
    """Return the average price of an item from the menu.

    >>> average_menu_price({'fries': 4.0, 'hamburger': 6.0})
    5.0
    """
    prices = [menu[item] for item in menu]
    return sum(prices) / len(prices)
```

Indeed, you have performed remarkably complex computations up to this point using just comprehensions to filter and transform data, and Python's built-in functions to aggregate

this data. For loops provide an alternate approach to these comprehensions that offer a trade-off of *code complexity* vs. *flexibility*. Comprehensions and built-in functions are often shorter and more direct translations of a computation than for loops, but for loops allow us to customize exactly how filtering and aggregation occurs.[6]

[6] A good rule of thumb to follow in this course is to use comprehensions and built-in functions when possible, and use loops when you really need a custom aggregation.

Of course, on your journey to learning programming it is important that you learn and master both of these techniques, and be able to translate between them when possible! Just as there are many ways to visualize a sunset (a painting, a photograph, a drawing, pixel art), so too are there many ways to implement a function. So whenever you see some code for a function involving comprehensions or loops, remember that you can always turn it into an additional learning opportunity by trying to rewrite it with a different approach.

References

- CSC108 videos: For loop over str (Part 1, Part 2, Part 3)